Software Model Checking for GPGPU Programs, Towards a Verification Tool

Unmesh Bordoloi
Linköping University
SE-581 83, Sweden
unmesh.bordoloi@liu.se

Ahmed Rezine
Linköping University
SE-581 83, Sweden
ahmed.rezine@liu.se

ABSTRACT

The tremendous computing power GPUs are capable of makes of them the epicenter of an unprecedented attention for applications other than graphics and gaming. Apart from the highly parallel nature of the programs to be run on GPUs, the sought after gain in computing power is only achieved with low level tuning at threads level and is therefore very error prone. In fact the level of intricacy involved when writing such programs is already a problem and will become a major bottleneck in spreading the technology.

Only very recent and rare works started looking into using formal methods for helping GPU programmers avoiding errors like data races, incorrect synchronizations or assertions violations. These are at their infancy and directly import techniques adapted for other (sequential) systems with simple approximations for concurrency [6]. Besides that, the only help we are aware of right now [4] takes a concrete input and explores a tiny portion of the possible thread scheduling looking for such errors. This easily misses common errors and makes of GPU programming a nightmare task. There is therefore still a lot of work to do in order to come up with helpful and scalable tools for today's and tomorrow's GPGPU software.

We state in this paper our intention in building in Linköping a flagship verification tool that will take CUDA code and track and report, with minimal assistance from the programmer, errors like data races, incorrect synchronizations or assertions violations. In order to achieve this ambitious and vital goal for the widespread of GPU programming, we build on our experience using and implementing CUDA and GPU code and on our latest work in the verification of multicore and concurrent programs. In fact, GPU programs like those written in CUDA are suitable for verification as they typically neither manipulate pointer arithmetics nor allow recursion. This restricts the focus to concurrency and array manipulation, combined with intra and inter procedural analysis. To give a flavor of where we start from, we report on our experiments in automatically verifying two synchronization algorithms that appeared in a recent paper [7] proposing efficient barriers for inter-block synchronization. Unlike any other verification approach for GPU programs, we can show that the algorithms neither deadlock nor violate the barrier condition regardless of the number of threads. We also capture bugs in case basic relations are violated between the number of blocks and the number of threads per block.

Categories and Subject Descriptors
D.2.4 [**Software/Program Verification**]: Model Checking, Formal Methods

General Terms
Assertion Checkers, Verification

Keywords
GPU, Software Model Checking, CUDA, Formal Verification

1. CUDA PROGRAMMING MODEL

GPUs are used to accelerate parallel phases in modern programs. Typically these phases deal with data intensive operations. However, they are also more and more used for more general computing, like by exploring parallelization possibilities in dynamic programming. As an example of a GPU programming model, CUDA extends ANSI C and uses kernel functions to specify the code run by all threads in a parallel phase. This is an instance of the Single Program Multiple Data (SPMD) programming style. When kernel functions are launched, threads are executed in a grid partitioned into a number of blocks. More precisely, executing a kernel function results in a one, two or three dimensional grid consisting of a number of blocks each having the same number of threads. Each thread can obtain the identifier of its block by using CUDA supplied variables (blockIdx.x, etc). Threads in the same block share a low latency memory. Those belonging to different blocks would need to use a much slower memory or to pass through the host. In addition to block identifiers, a thread has also identifiers it can access using other CUDA variables (threadIdx.x, etc). Based on these indices, each thread will run the kernel function differently as the latter can refer to them.

2. SYNCHRONIZATION EXAMPLES

In the following, we describe two synchronization barriers from [7]. The two solutions propose inter-block barrier im-

```
1 //blocks shared variable
2 __device__ int g_mutex;
3
4 //centralized barrier function
5 __device__ void __gpu_sync(int goalVal){
6     int tid_in_block = threadIdx.x * blockDim.y +
7        threadIdx.y;
8
9     // in each block, only thread 0 synchronizes
10    // with the other blocks
11    if (tid_in_block == 0) {
12       atomicAdd(&g_mutex, 1);
13       // wait for the other blocks
14       while(g_mutex != goalVal) { }
15    }
16
17    // synchronize within the block
18    __synchthreads();
19 }
```

Figure 1: Code snapshot of a simple barrier [7].

plementations. Indeed, CUDA only supplies intra-block synchronization barriers using the "__synchthreads()" function. The authors in [7] propose a centralized mutex based solution together with a more efficient "lock-free" solution. In fact they also propose a tree based solution that can be regarded as a direct extension of the simple solution. For lack of space, we concentrate here on the two first solutions.

Centralized simple Synchronization. A snapshot of the code for this solution is presented in Figure 1. When calling the "__gpu_sync" function implementing the inter-block barrier, the number of blocks is passed as the value "goalVal". The idea is to use a global variable shared by all blocks, here "g_mutex". The solution assumes a "leading thread" in each block. After the block completes its computation in the current epoch, its leading thread atomically increments the shared variable "g_mutex" (line 12). The leading thread starts then its active wait (line 14) until the variable evaluates to "goalVal", in which case the leading thread can synchronize with the other threads in its block, hence proceeding to the next epoch.

Lock free synchronization. A snapshot of this second solution is described in Figure 2. Instead of having all blocks accessing the same shared variable, this solution proposes to share two arrays (namely "Ain" and "Aout") with the number of blocks as their respective sizes. The idea is that each block that completed the computation in the current epoch will have its leading thread assign "goalVal" (passed as a parameter together with the arrays) to the input array "Ain" in the slot corresponding to its block. After that, the leading thread waits for the slot corresponding to its block in "Aout" to become "goalVal". After all blocks have assigned "goalVal" to their respective slot in "Ain", the threads of a chosen block (here block 1) synchronize at line 25. The threads belonging to the chosen block are used to monitor, for each block id, "Ain[id]" and to assign its value to "Aout[id]" in case it evaluates to "goalVal". As a result, all leading threads can proceed and synchronize with the threads in their own block (line 38) hence moving to the next epoch.

```
1 //lock-free barrier function
2  __device__ void __gpu_sync(int goalVal, int *Ain,
3                               int *Aout){
4
5     int tid_in_block = threadIdx.x * blockDim.y +
6           threadIdx.y;
7     int nBlockNum = gridDim.x * gridDim.y;
8     // each thread of the first block is
9     // associated to a block it monitors
10    int bid = blockIdx.x * gridDim.y + blockIdx.y;
11
12    // thread 0 of each block states its arrival
13    if (tid_in_block == 0){
14        Ain[bid] = goalVal;
15     }
16
17    if (bid == 1){
18        // threads in the first block wait for
19        // the associated block
20        if (tid_in_block < nBlockNum) {
21           while(Ain[tid_in_block] != goalVal){ }
22        }
23
24        // synchronize threads of first block
25         __synchthreads();
26
27          // release associated block
28        if (tid_in_block < nBlockNum) {
29            Aout[tid_in_block] = goalVal;
30        }
31    }
32    // each block waits to be released
33    if (tid_in_block == 0) {
34        while(Aout[tid_in_block] != goalVal) { }
35    }
36
37    // synchronize within the block
38      __synchthreads();
39 }
```

Figure 2: Code snapshot of a lock-free barrier [7].

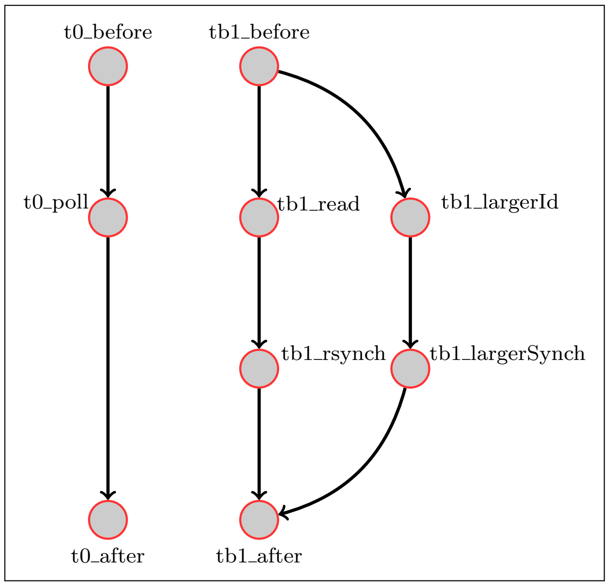

Figure 3: Model of the lock free barrier of Fig.2. Initially, all leading threads are in "t0_before" and all block 1 threads are in "tb1_before".

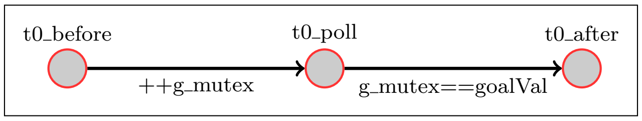

Figure 4: Model of the simple barrier in Fig. 1. Initially, all leading threads are in state "t0_before".

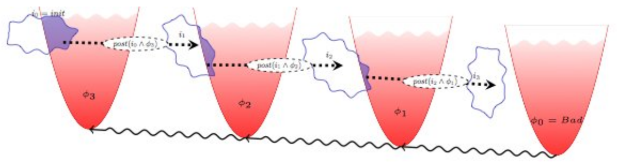

Figure 5: Counter Example Guided Abstraction Refine. for Concurrent Parameterized Systems [2]

3. PARAMETERIZED VERIFICATION

Given models of the programs, we perform automatic parameterized verification, i.e., we verify with minimal human interaction programs regardless of the number of concurrent threads and blocks in the system. The problem can be shown to be undecidable in general and combinations with abstractions and efficient symbolic representations play an important role. For this reason, we build on our previous work with *monotonic abstraction* [3] and its automatic refinement [2]. Monotonic abstraction is based on the concept of *monotonic systems w.r.t. a well-quasi ordering* $\preceq$ defined on the set of configurations [1, 5]. Since the abstract transition relation is an over-approximation of the original one, proving a safety property in the abstract system implies that the property is also satisfied in the original system. However, this also implies that *false-positives* may be generated in the abstract model, we handle them by combining forward/backward analysis together with widening or interpolation techniques. This is schematically described in Figure.3. More details are available in [2].

4. EXPERIMENTS AND FUTURE WORK

At this stage, we manually build the models in Figures 4 and 3 to describe behaviors of the programs in Figures 1 and 2. Of course, building such models is both time consuming and error prone. We are working on automatically extracting such models from CUDA source code without the need to manually supply them. This model extraction step is also combined with techniques like slicing or predicate abstraction to boost the applicability and the scalability of the approach. Assuming for now the models are given, our verification technique applied to the first simple synchronization algorithm, captures that: if "goalVal = nBlocsNum" then the algorithm of Figure 1 respects the barrier property and does not deadlock. This is not the case if "goalVal<nBlocksNum" (barrier property violated) or "goalVal>nBlockNum" (deadlock). For the algorithm of Figure 2, our prototype automatically captures that if "nThreadsPerBlock $\geq$ nBlocksNum", then the algorithm respects the barrier property, and does not otherwise.

Model	pass	seconds
CS[7]: goalVal = nBlocksNum	$\surd$	0.05
CS[7]: goalVal < nBlocksNum	$\times$	<0.01
CS[7]: goalVal > nBlocksNum	$\times$	0.05
LFS[7]: nThreadsPerBlock $\leq$ nBlocksNum	$\surd$	2.7
LFS[7]: nThreadsPerBlock < nBlocksNum	$\times$	<0.01

Table 1: We use our prototype for automatic parameterized verification [2]. $\surd$ stands for verified, and $\times$ for supplying a concrete counter example.

These results are relevant as a typical number of threads per block on the latest generation of GPUs is 32. This is the same number as the size of the "warp" (often selected by the designers to hide latencies). A warp can be seen as the smallest unit of threads that are scheduled together by a GPU multiprocessor. Hence, choosing this number as the thread block size has clear advantages. In future GPUs, we can expect this number to remain more or less the same. On the other hand, the number of multiprocessors on GPUs can be expected to increase rapidly, as has been the trend. For example, nVIDIA Tesla M2050 already has 14 multiprocessors, and in future we can expect this number to be easily more than the magic number 32. Given this and given the synchronization approach in the previous algorithms, we can easily have more thread blocks than threads per block. These algorithms would therefore not respect the barrier property if ported directly to other platforms. This was a simple example showing that capturing such errors and proving their absence while allowing for benign race conditions or other performance tricks is very important as it helps programming GPU platforms.

5. REFERENCES

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96, 11th IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.

[2] P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine. Constrained monotonic abstraction: A cegar for parameterized verification. In *CONCUR 2010 - Concurrency Theory, 21th International Conference*, pages 86–101, 2010.

[3] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007*, pages 721–736, 2007.

[4] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Progr. In *3rd Workshop on Software Tools for MultiCore Systems*, 2008.

[5] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Comput. Sci.*, 256(1-2):63–92, 2001.

[6] G. Li, G. Gopalakrishnan, R. M. Kirby, and D. Quinlan. A symbolic verifier for cuda programs. *SIGPLAN Not.*, 45:357–358, January 2010.

[7] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.